SUPLEMENTO MENSUAL DEDICADO A LA IMAGEN SINTÉTICA
Número 21

Taller Virtual
Presentamos las nuevas sentencias de programación

Cómo...
Preparar animaciones con POV

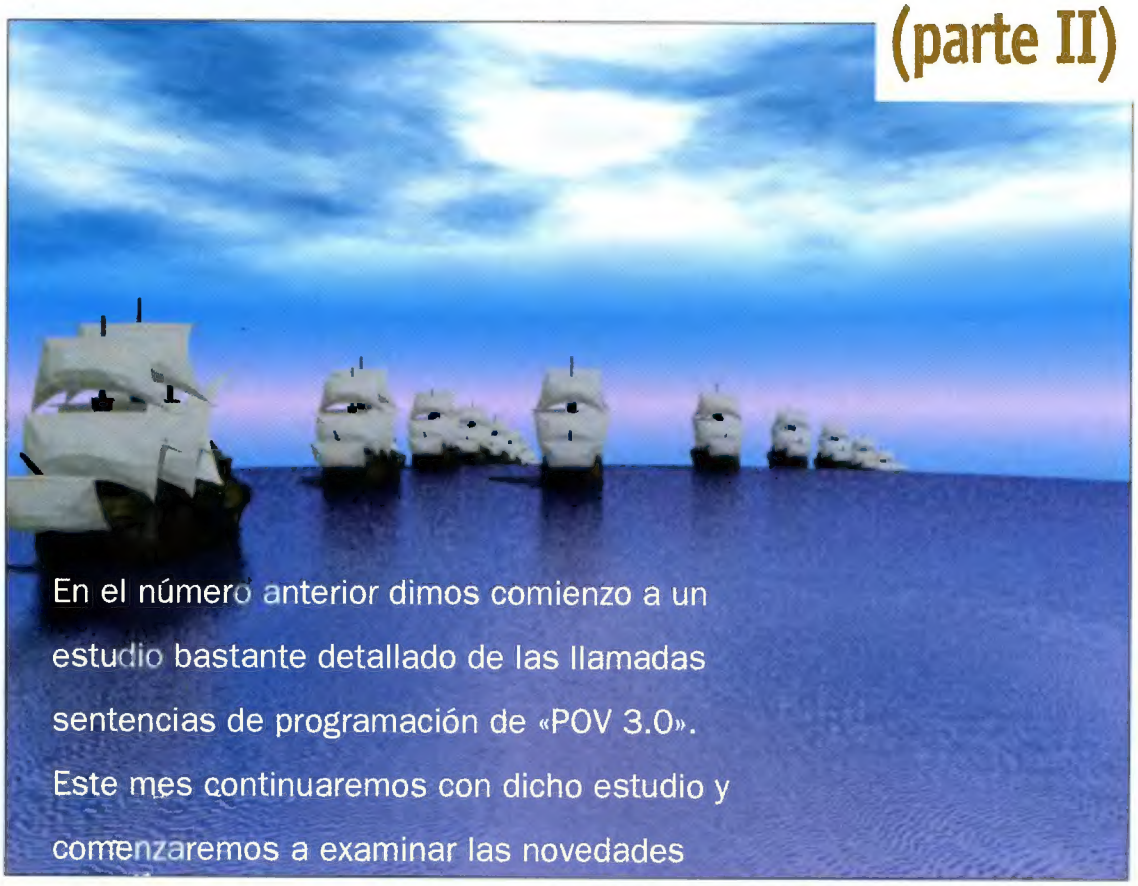
El foro del lector
Animaciones de gran tamaño

render MANIA

pcmania

José Manuel Muñoz

Sentencias de programación (parte II)



En el número anterior dimos comienzo a un estudio bastante detallado de las llamadas sentencias de programación de «POV 3.0». Este mes continuaremos con dicho estudio y comenzaremos a examinar las novedades que en este apartado han aparecido con la versión beta de «POV 3.1». Estas novedades van a suponer una ayuda extraordinaria para los povmaníacos en general y para los escritores de pov-ipas en particular, y por ello recomendamos su aprendizaje a todos los pov-adeptos.

Aunque quienes tengan conocimientos de programación no tendrán ninguna dificultad en digerir estos artículos, hay muchos lectores que no se hallan en este caso. Por esta razón se ha procurado que las explicaciones y ejemplos resultasen tan claros como fuese posible. Hay que hacer hincapié, sin embargo, en que los experimentos propios y la práctica son el único camino para do-

minar estas sentencias. Así pues, después de leer los artículos (y mirar los ficheros de ejemplo), el lector debería realizar sus propias pruebas y no dejarse descorazonar si comete errores o si las cosas no salen bien a la primera. Sólo la práctica proporciona, en la programación, la habilidad necesaria para acometer diferentes proyectos complejos. Dichas sentencias son prácticamente imprescindibles para:

1) Crear más fácilmente ciertos objetos cuya construcción "manual" puede resultar muy fastidiosa.

2) Componer escenas formadas por cientos o miles de modelos complejos (importados o no).

3) Preparar las diferentes animaciones complicadas.

4) Escribir pov-ipas de todo tipo.

5) Crear posturas aleatorias para diferentes modelos.

Utilizando rotate podemos situar los galeones sobre un mundo de cualquier tamaño.

Rand, Seed y otras funciones

En el número anterior aprendimos cómo podía utilizarse la sentencia #while para situar en una escena a cientos o miles de objetos escribiendo tan sólo unas pocas líneas en el fichero escénico. Esta sentencia es la base de las escenas de batallas medievales entre orcos y humanos que se publicaron en el número 17 de Rendermanía (y de otras muchas imágenes). Sin embargo, el uso de #while por sí solo será insuficiente en ciertos casos. Supongamos, por ejemplo, que deseamos crear una escena en la que hay un regimiento de caballeros medievales. Podemos crear dicho regimiento utilizando un doble bucle #while anidado de la manera que vimos en el artículo anterior. De esta forma podremos crear una formación con un número cualquiera de filas y columnas. El problema es que el efecto resultante será de una perfección excesiva: todos los caballeros estarán alineados formando un cuadro tan perfecto que la escena no resultará demasiado creíble. Pero, afortunadamente, podemos resolver este problema empleando las funciones rand y seed.

¿Que de dónde salen estas funciones? «POV» dispone de una serie de funciones para manipular valores en flotante, vectores y strings (cadenas de texto), y rand y seed están entre las funciones del primer tipo. Estas funciones aceptan como entrada uno o dos parámetros en flotante (que deben escribirse entre paréntesis) y devuelven otro valor en flotante como salida. La siguiente línea sería un ejemplo válido de uso de seed

```
#declare R1=seed(16384);
```

Con esta línea estaremos abriendo un canal llamado R1 para utilizar el generador de números pseudoaleatorios de «POV». Después de esto, cada vez que invoquemos a rand dándole R1 como parámetro, esta función nos devolverá el siguiente número pseudoaleatorio para este canal. Así, por ejemplo, con la línea...

```
#declare A=rand(R1);
```

la variable A tomará un valor pseudoaleatorio devuelto por la función rand. Este valor estará siempre comprendido dentro del rango entre 0 y 1 (incluyendo ambos valores), y si escribimos más líneas como la anterior, obtendremos siempre valores diferentes para A.

¿Qué quiere decir todo esto? Comenzaremos aclarando que un generador de números pseudoaleatorios es un algoritmo que emplea una fórmula dada para producir una larga secuencia de números. Estos números son, aparentemente, aleatorios, pero en realidad es fácil producir la secuencia si conocemos la fórmula que se está empleando. El algoritmo acepta un valor como semilla y este valor se emplea para decidir el punto de la secuencia desde donde rand empezará a devolver valores al usuario. Esto quiere decir que una semilla dada producirá siempre la misma

Ejemplos del uso de rand

```
...
...
object{#include "soldado.inc" texture{pielorco}
    #declare alto=.8 + rand(R)*.25;
    #declare ancho=.90 + rand(R)*.2;
    scale <1*ancho,1*alto,1*ancho>
    #declare sumx=rand(R)*30;
    #declare sumz=rand(R)*30;
    translate <Xbase+xpos+sumx, 0, Zbase+zpos+sumz>
}
```

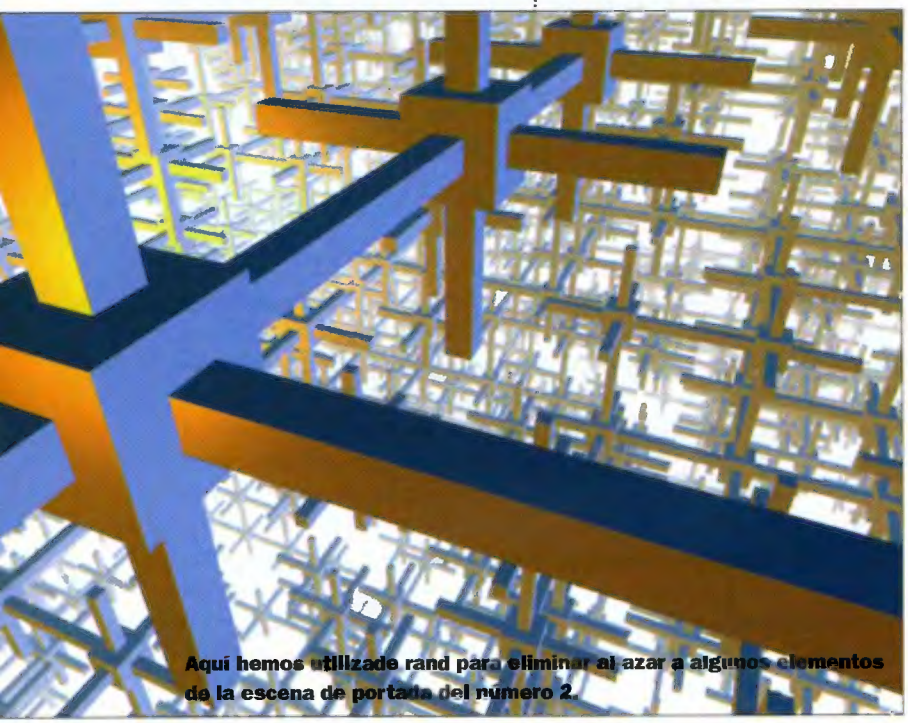
Figura 1

```
#declare R1 = seed(16384);
#declare R2 = seed(12345);
sphere { <rand(R1), rand(R1), rand(R1)>, rand(R2) }
```

Cuadro1

secuencia de números. En fin, volviendo al ejemplo del regimiento, echad un vistazo a la **Figura 1.**

Este ejemplo está tomado de uno de los ficheros utilizados para las escenas de batallas, y en él la sentencia object empleada está incluida dentro del bucle #while doble empleado para construir un regimiento. Las variables Xbase, xpos, Zbase y zpos se han empleado para situar a cada soldado dentro de su posición ideal dentro del cuadro. Pero aquí no nos interesa cómo se han calculado estas variables. Sólo nos interesa saber que sumx y sumz son variaciones aleatorias que se suman a la posición ideal de cada soldado, para conseguir que la formación no resulte excesivamente perfecta. Tomad nota de que el valor devuelto por rand sigue estando entre 0 y 1 pero, al multiplicarlo por 30, conseguimos que los valores para sumx y sumz oscilen entre 0 y 30 unidades. Además, hemos empleado rand para crear pequeñas variaciones aleatorias en la estatura y el ancho de cada soldado. Para la estatura, por ejemplo, se multiplica el valor devuelto por rand por 0.25 (con lo que éste oscilará entre 0 y 0.25) y luego se le suma el valor 0.8. Con ello el valor resultante estará siempre comprendido entre 0.8 y 1.05.



Aquí hemos utilizado rand para eliminar al azar a algunos elementos de la escena de portada del número 2.

De esta forma, en la sentencia denominada scale, la altura del personaje se multiplicará por la variable alto, con lo que el personaje encogerá o crecerá según el valor de esta variable.

Por otro lado, puede suceder que el usuario desee emplear varias semillas simultáneamente y por esta razón, en «POV», el generador funciona a través de "canales". En el manual tenemos el ejemplo representado en el **Cuadro 1**.

Con estas líneas se crea una esfera cuya posición en los ejes X,Y y Z se define con los valores devueltos por rand con el canal R1. En cambio, el radio de la esfera depende del valor devuelto mediante el canal R2. Pero ¿para qué puede servirnos el emplear más de un canal para el generador? Siguiendo con el ejemplo del regimiento, imaginemos que se ha empleado rand para definir las posiciones de cada soldado y para dar valores de rotación aleatorios a los brazos y piernas de cada modelo, definiendo así una postura distinta para cada uno (cosa que ya hemos hecho en números anteriores). Está claro que, como cada semilla produce siempre la misma secuencia de números, una misma semilla producirá siempre la misma escena. Pero ahora supongamos que decidimos, además, utilizar rand para crear leves variaciones en la pigmentación de la armadura de cada soldado (haciendo sumas aleatorias en los valores originales de rgb de cada uno). Si empleamos un sólo canal para todo esto, el regimiento que obtendremos será distinto. ¿Por qué? Porque los valores devueltos por rand, y que antes sólo se empleaban en la posición y rotación de los modelos, se emplearán ahora también para alterar

Hemos utilizado un array para crear la formación de la flota (si=1, hay barco)

el color de éstos. Así pues, como ahora cada soldado necesitará más valores rand, a partir del segundo soldado la secuencia de valores rand devueltos estará desplazada con respecto a la escena anterior. Obtendremos, así, valores distintos para las variaciones aleatorias de posición y escalado y también para la definición de posturas, y por ello la nueva escena será completamente distinta a la anterior. Sin embargo, si deseamos generar la misma escena de antes (salvo por los cambios en el color), podremos hacerlo utilizando el mismo canal de antes para las tareas de variación posicional y definición de posturas, al tiempo que empleamos otro nuevo que sólo será referenciado por rand en las cuestiones de variación del color de las armaduras.

Resumiendo, las funciones rand y seed pueden ser empleadas para producir variaciones posicionales. de escala. de orientación, de cambios de color, y de elección de modelos, por citar sólo algunas de sus posibilidades.

Más funciones flotantes

A continuación, señalaremos algunas de las funciones que tratan valores en flotante y que se utilizan con mayor frecuencia:

abs(A): Retorna el valor absoluto de la variable A. Si A era negativa, el valor devuelto lo será también.

asc(S): Devuelve un valor entero comprendido entre 0 y 255. Este valor es el que corresponde al primer carácter ASCII de la cadena S dada como entrada. Por ejemplo asc("ABC") devuelve 65 por "A".

ceil(A): Retorna el entero más pequeño mayor que A.

cos(A): Retorna el coseno de A. Tanto A como el valor devuelto se dan en radianes.

degrees(A): Devuelve en grados el valor de A (que estaba en radianes).

div(A/B): Retorna la parte entera de A/B.

floor(A): Devuelve el mayor entero menor que A.

int(A): Trunca la parte decimal de A y devuelve sólo el valor entero.

max(A,B): Retorna A o B dependiendo de cuál de los dos tenga el mayor valor.

min(A,B): Devuelve A o B según cuál de los dos sea el valor menor.

mod(A/B): Retorna el modulo de A/B, o sea el resto después de la división.

radians(A): Convierte un valor dado en grados a radianes.

sin(A): Devuelve el seno del ángulo A.

tan(A): Devuelve la tangente del ángulo A.

val(S): Retorna el valor en flotante correspondiente a la cadena suministrada. Por ejemplo val("1234.56"), nos devolverá el valor 1234.56.

Existen algunos casos en los que es indispensable el uso de algunas de estas funciones. Por ejemplo, supongamos que hemos creado tres modelos diferentes de casas y deseamos generar un pueblo alternando las casas aleatoriamente. Para lograr la escena podemos utilizar un doble bucle para posicionar las casas y a la función seed para alterar un poco la colocación. También podemos utilizar rand para decidir qué casa poner antes de escribir la sentencia object. Ahora bien, hay que recordar que rand siempre devuelve valores en flotante entre 0 y 1. ¿Cómo podemos escribir esto? Una posible solución la podemos encontrar en el trozo de código indicado en la **Figura 2.** Como puede verse en este ejemplo, int

Uso de rand e int para elegir objetos aleatoriamente

```
...
...
#declare cual=int(rand(S)*3);
#if (cual=0)
      object{casa1 translate<posx, posy, posz>}
#end
#if (cual=1)
      object{casa2 translate<posx, posy, posz>}
#end
#if (cual=2)
      object{casa3 translate<posx, posy, posz>}
#end
...
...
```

Figura 2

trunca la parte decimal del valor devuelto por rand, con lo que la variable "cual" valdrá siempre 0, 1 o 2. Por supuesto, también podríamos evitar el uso de int y cambiar las condiciones escritas en los #ifs, pero, tal y como está, el código resulta más sencillo de leer. El resultado, no obstante, no es demasiado elegante y lo sería aún menos si tuviéramos 20 modelos de casa entre los que elegir. En este caso podríamos escribir 20 #ifs, pero sería mucho mejor utilizar otra sentencia condicional: #switch.

#SWITCH

En la **Figura 3** tenemos un ejemplo de cómo podríamos utilizar #switch en el caso del ejemplo de las 20 casas. En primer lugar, debemos situar la sentencia en el centro del doble bucle #while (o de cualquiera que sea el trozo de código que estemos empleando para dar valores a las variables de posición de los objetos). Posteriormente, observad la primera sentencia del ejemplo. Como véis, el valor que multiplica al devuelto por la función rand no es 20 sino 40. ¿Por qué? Porque hemos decidido complicar un poco el ejemplo de las 20 casas. En efecto, seguimos teniendo 20 modelos de casas, pero ahora queremos que dos de ellas se utilicen con más frecuencia en la construcción del pueblo. Veamos cómo funciona esto. En primer lugar debemos colocar la variable "cual" dentro de los paréntesis que siguen a la sentencia #switch. Así, esta variable será empleada para estudiar las condiciones de la sentencia. El primer ejemplo de esto lo tenemos en la siguiente línea. En ella se abre con #case un caso que se cumplirá si el valor de la variable "cual" corresponde al que hay indicado entre los paréntesis del #case. Si se cumple este caso, entonces se compilarán todas las sentencias que hayamos escrito entre el #case y la sentencia #break. (En cuanto a la función de esta última, su propósito es, por supuesto, marcar el final del bloque de sentencias abierto con la sentencia #case). Así pues, si "cual" es igual a 0, se creará el objeto casa1 y, al encontrar el #break, POV saltará al final de la sentencia #switch (dicho final está marcado por la sentencia #end, tal y como sucede con #if y #while).

Naturalmente, en caso de que "cual" valiese 1, las sentencias de este primer caso no se compilarían, haciéndolo en su lugar las incluidas dentro del bloque delimitado por el segundo #case (con lo que se crearía el objeto casa2). Podemos escribir tantos bloques #case como queramos dentro de una sentencia #switch, y dentro de cada bloque #case puede haber tantas sentencias como sea preciso (incluso podemos tener dentro otras sentencias #switch, largos bucles #while, macros, o lo que sea). Sin embargo, y en aras de la claridad, no es recomendable que estos bloques sean excesivamente largos. Además, hay que recordar que dentro de los paréntesis de las sentencias #case deberemos escribir siempre constantes o valores numéricos, nunca variables.

Ahora veamos cómo funcionan otras dos sentencias asociadas a #switch. La primera de ellas es #range y se emplea para indicar que el bloque de sentencias delimitado por ella misma y por #break se compilará sólo si el valor dado al #switch está comprendido entre los dos valores indicados entre los paréntesis de #range o es igual a alguno de dichos valores. En el ejemplo, estos dos valores (que deben estar separados por una coma) son 20 y 30, de manera que, como los posibles valores dados a "cual" estarán siempre entre 0 y 39, esto quiere decir que aproximadamente una de cada cuatro casas del pueblo serán del modelo casa19.

Por último, tenemos el caso del #else que, como ya vimos con la sentencia #if, puede traducirse literalmente como "y si no...". Cuando «POV» encuentre esta sentencia en el lugar de #case o #range, entonces se compilará su bloque de sentencias si ninguno de los casos expuestos por las sentencias #case o #range se cumple.

Resumiendo, #switch es una sentencia de decisión utilizada para casos de ramificaciones múltiples en los que el uso de #if, aunque posible, es desacon-

Ejemplo de uso de #switch y sus sentencias asociadas"

```
...
...
#declare cual=int(rand(S)*40);
#switch (cual)
        #case (0)
        object{casa1 translate<posx, posy, posz>}
        #break
        #case (1)
        object{casa2 translate<posx, posy, posz>}
        #break
        ...
        ...
        #range(20,30)
        object{casa19 translate<posx, posy, posz>}
        #break
        #else
        object{casa20 translate<posx, posy, posz>}
#end
...
...
```

Figura 3

En la portada del número 0 se utilizó un #switch con unos 28 cases que testeaban el valor devuelto por rand para cada casa.

sejable debido a la dificultad de lectura que comportaría la escritura de muchos #ifs encadenados. Por otro lado, con #switch sólo podemos establecer comparaciones de igualdad con la variable escrita entre los paréntesis de la sentencia (salvo por el uso de #range). Con #if, en cambio, podemos establecer condiciones mucho más complicadas. Un buen ejemplo de #switch fue la portada del primer número de Rendermanía, donde se utilizó esta sentencia para crear (empleando, también, #while y rand) un pueblo con diferentes casas medievales.

Arrays

Hasta ahora todas las sentencias de "programación" explicadas son las que aparecieron con «POV 3.0». A partir de este momento, comenzaremos a estudiar las que añade la nueva versión 3.1. Una de ellas es array, una sentencia que sirve para definir estructuras de datos que se conocen precisamente con ese nombre: arrays. (Otros nombres menos utilizados son arreglos, matrices y tablas). De nuevo este nombre significará mucho para cualquier programador y efectivamente los arrays de POV 3.1 se parecen bastante a los del lenguaje C. Pero, ¿qué es un array?

Un array es una agrupación de variables del mismo tipo a las que se llama por un mismo nombre. Se accede a cada elemento del array especificando el nombre del array y añadiendo un índice a dicho nombre. Veamos un ejemplo sencillo de cómo se crea un array:

```
#declare misdatos = array[10]
```

Esta sentencia crea el array de una sola dimensión llamado misdatos que consta de diez elementos. El primer elemento de un array de este tipo se referencia siempre con el valor 0 para su índice, por lo que es fácil imaginar que, en este caso, el último elemento de nuestro array será referenciado con el valor 9. Así, la sentencia:

```
#declare misdatos[0]=56
```

guarda el valor 56 en el primer elemento de la tabla mientras que la siguiente sentencia:

```
#declare yuyu=misdatos[0]
```

copiará el valor 56 guardado en este elemento en la variable yuyu.

Ahora conviene hacer unas puntualizaciones. A diferencia de como sucede con los arrays del lenguaje C, en «POV» podemos crear un array sin indicar el tipo de datos que va a guardar. Esto es lo que hemos hecho al crear el array misdatos. Sin embargo, en cuanto asignemos un tipo de dato a cualquier elemento del array, todos los demás sólo podrán guardar ese tipo de dato. En el caso anterior hemos asignado un valor numérico al primer elemento del array, por lo que POV nos dará un error (y dejará de compilar) si intentamos asignar (por ejemplo) una textura a otro elemento del mismo. Ahora bien. esto nos conduce a una sencilla pregunta; ¿qué tipo de datos podemos guardar en los arrays de «POV»? Pues sencillamente cualquier cosa que pueda ser

Algunos ejemplos de uso de arrays

```
...
...
//ejemplo 1: las siguientes lineas no crean ningun objeto
#declare misdatos = array[10]
#declare misdatos[5] = 57;
//la siguiente sentencia falla porque ya se ha establecido que misdatos va a guardar valores numericos
#declare misdatos[0]=pigment{White}      //esta sentencia provoca un fallo de compilación
//La sentencia que sigue tambien dara error porque el elemento misdatos[4] aun no ha sido inicializado
#declare bill = misdatos[4];
#declare misdatos[3]=misdatos[5]+92;      //en cambio esta sentencia funcionara


//ejemplo 2: crea una esfera blanca sobre fondo verde
#declare colorines = array[3]{White, Red, Green}
#declare tprueba1=texture{pigment{colorines[0]} finish{F_MetalA}}
#declare tprueba2=texture{tprueba1 finish{ambient 0.16}}
sphere{<0,20,0>,20 texture{tprueba1}}
plane{y,0 pigment{colorines[2]}}


//ejemplo 3: crea una esfera roja sobre un plano verde
#declare colorines = array[3]{White, Red, Green}
#declare tprueba1=texture{pigment{colorines[0]} finish{F_MetalA}}
#declare yuyu=array[2]
#declare yuyu[0]=sphere{<0,20,0>,20 texture{tprueba1}}
#declare yuyu[1]=plane{y,0 pigment{colorines[2]}}
object{yuyu[0] pigment{Red]}
object{yuyu[1]}
...
...
```

Figura 4

asignada con una sentencia #declare. Así pues, aparte de valores numéricos, podemos usar arrays para guardar vectores, colores, normales, acabados, definiciones completas de texturas e incluso objetos.

Por otro lado, como el usar sentencias #declare para inicializar cada elemento de un array sería algo bastante pesado, «POV» nos ofrece otra forma de hacerlo. Por ejemplo, la siguiente sentencia nos servirá para inicializar todos los elementos de misdatos al mismo tiempo que creamos este array.

```
misdatos=array[10]{57,1,2,3,4,5,8342,45,0,12}
```

Con esto, misdatos[0] guardará el valor 57, misdatos[1] guardará el valor 1 y así sucesivamente. Este mismo sistema se puede utilizar con otros tipos de datos. Por ejemplo, con colores:

```
misdatos=array[3]{Red, White. Green}
```

Naturalmente, para que este último array sea válido, los identificadores de los colores habrán de estar inicializados (lo que podemos hacer cargando previamente el fichero colors.inc). Ahora echad un vistazo a la **Figura 4** donde hemos puesto algunos ejemplos correctos e incorrectos de creación y uso de arrays.

En el primer ejemplo, la asignación de un dato de tipo pigmento a un elemento de un array en el que ya se había asignado un valor numérico provoca un error. (De hecho, esta línea provoca un cuelgue de POV en la versión beta con la que se han realizado estas pruebas. Es de suponer que esto no sucederá en la versión definitiva). La asignación a la variable bill también falla porque el elemento misdatos[4] está sin inicializar. En cambio, la asignación del valor para misdatos[3] funciona.

El ejemplo 2 es un poco más complejo. Primero se define un array de colores y luego se usa uno de los elementos de dicho array en la asignación del pigmento de la textura tprueba1. Luego, a su vez, esta textura es utilizada normalmente dentro de la declaración de un objeto -una esfera- mientras que otro elemento del array es empleado directamente como pigmento dentro del objeto plane. Este ejemplo se renderizará sin fallos y su resultado será una esfera blanca situada sobre un plano verde.

Hasta ahora creábamos estas formaciones con bucles dobles (#while) pero el uso del array nos da mayor flexibilidad.

Y por último tenemos un tercer ejemplo que es un pequeño refinamiento sobre el anterior. Como antes, se crea un array de colores y se usa uno de sus elementos -el cero. que tiene asignado el color blanco- para declarar la textura tprueba1. Luego se declara un nuevo array llamado yuyu y se le asigna como primer elemento una esfera sobre la que se aplica la textura tprueba1. Después se inicializa el siguiente elemento de este array, asignándole un plano sobre el que se aplica como pigmento otro de los elementos del array de colores. Por último, creamos los dos objetos del array yuyu, para lo cual únicamente tenemos que referenciar a estos dos elementos desde dentro de sendas sentencias object. Este ejemplo también compilará sin fallos y el resultado será una esfera roja sobre un plano verde. (Nota: si queréis comprobar el funcionamiento de estos ejemplos –que se hallan en el fichero arraypru.pov– debéis probarlos uno a uno, desconectando –con comentarios– todos los demás en cada prueba). La esfera es roja porque aunque yuyu[0] es una esfera blanca, el pigmento del objeto ha sido sustituido por otro al invocar a dicho objeto.

En resumen, recordad que:

1) Una vez que se asigna un elemento de un tipo dado a un array, todos sus demás elementos deben de ser del mismo tipo.

2) En un array cualquiera, no debemos referenciar elementos que no han sido inicializados.

3) Para emplear los elementos de un array, debemos referenciar a dichos elementos desde los mismos puntos en los que hubiéramos escrito el tipo de dato que contienen. Si el array guarda colores, desde dentro de un pigmento, si guarda texturas, desde dentro de una sentencia texture, si guarda objetos, desde dentro de una sentencia object, etcétera.

Arrays de más de una dimensión

Hasta ahora todos nuestros ejemplos han sido arrays de una dimensión. Estos son, como hemos visto, listas de datos que se guardan en posiciones contiguas de memoria, en orden creciente (0 para el índice al primer elemento, 1 para el segundo, etc.). Pero «POV» permite utilizar arrays multidimensionales de hasta un límite de cinco dimensiones, de los cuales el caso más sencillo es el del array de dos dimensiones. Un array de este tipo puede definirse como un array de arrays de una dimensión y su formato queda así:

```
#declare nombre_array=array[tamaño_primera
_dimensión][tamaño_segunda_dimensión]
```

Así pues, para declarar el array bidimensional yuyu de tamaño 4*3, escribiríamos:

```
#declare yuyu=array[4][3];
```

En cambio, para asignar un valor al primer elemento del array tendríamos que hacer lo siguiente:

```
#declare yuyu[0][0]=46;
```

Como puede verse, en estos arrays se utilizan dos índices para definir su tamaño y acceder a sus elementos. Los arrays bidimensionales se almacenan en matrices en las que el primer índice indica la fila y el segundo la columna. En la memoria del ordenador esto se almacena de forma que primero van las columnas y luego las filas. Se comienza, pues, con el elemento 0 de la primera columna, se continúa con el 0 de la siguiente y así hasta que se pasa a la primera columna de la siguiente fila. Esto es, el orden de almacenamiento hace que el índice de la derecha vaya cambiando más deprisa que el primero. Por otro lado, estos arrays pueden inicializarse también en el momento de ser creados, aunque el formato cambia un poco con recpecto a los arrays unidimensionales. Examinad la **Figura 5.**

En este ejemplo, el elemento [0][0] del array es inicializado con el valor 7, el [0][1] lo hace con el valor 6 y el [0][2] lo hace con el valor 4. Después, los siguientes elementos (si seguimos el orden de almacenamiento) serían el [1][0] (que vale 1), el [1][1] (que vale 2), el [1][2] (que vale 3), etc. Obsérve-

Declaración de un array bidimensional

```
//así pueden declararse el array yuyu de 4*3 elementos
#declare yuyu=array[4][3]
    {
    {7,6,4},
    {1,2,3},
    {0,9,8},
    {4,5,5}
}
```

Figura 5

se que en la creación de un array inicializado de este tipo hay que emplear comas para separar los diferentes elementos y las dimensiones (que se delimitan con llaves).

Los arrays de tres o más dimensiones, no suelen se utilizan muy amenudo, porque la cantidad de memoria que requieren crece de una forma exponencial con cada nueva dimensión.

Así por ejemplo, en el siguiente array reserva espacio para 4*6*8 elementos.

```
#declare yuyu=array[4][6][8]
```

Posibles usos de los Arrays

Aunque llevamos ya un buen rato hablando de arrays, aún no hemos dicho nada sobre su utilidad, de modo que vamos a reseñar una serie de posibles usos. Por supuesto, solo vamos a mencionar unos pocos ejemplos y será la práctica, en el futuro, la que nos permitirá hallar muchos más usos para esta nueva característica de «POV».

1) Suponed que hemos escrito un array bidimensional de valores numéricos donde cada número corresponde a la altura Y de un vértice de un objeto, el cual se usará como suelo en una escena en la que vamos a colocar personajes antropomórficos. Podríamos, entonces, al situar a cada personaje en dicho suelo, determinar el elemento de altura que le corresponde y asegurarnos que mantiene los pies en el suelo, sin que importen las irregularidades del mismo. ¡Se acabaron los suelos planos!

2) Imaginad que tenemos una superficie plana de forma irregular, como por ejemplo la cubierta de un buque de principios del siglo XIX. Queremos situar una serie de modelos en esta superficie de manera que no se superpongan unos y otros. Como dicha cubierta no tiene forma rectangular sería difícil, o imposible, utilizar #while para hallar los valores de posición de los modelos. Por ello, en vez de esto, podemos crear una gran matriz bidimensional para guardar valores numéricos que indiquen la distribución de los modelos sobre la cubierta. Un valor 0 podría significar ningún modelo, un valor 1 el modelo tal en tal postura, un valor de 2 otro modelo en otra postura, etc. Podríamos usar un doble bucle para leer el array y sería sencillo calcular la situación de cada modelo según la posición del valor leído en la matriz. (Según el tamaño de la superficie, podemos suponer que cada elemento esta a 50 unidades de los elementos adyacentes o a 300 o a 10000 unidades, según sea el caso).

3) Podemos usar un array para guardar listas de texturas y objetos simples y crear con ello un Pov-ipas de creación de árboles o de edificios o de lo que nos apetezca, con más facilidad.

4) Podemos guardar datos numéricos que se utilizarán en animaciones como trayectorias complejas de objetos, rotaciones, cambios de escala, cambios de cámara, etc.

En fin, se nos acabó el espacio. El mes que viene continuaremos tratando estas sentencias.

En estas pruebas –y en la propia portada–, se empleó el soldado de Óscar Alvárez Díaz.

Realizar animaciones con POV 3.1 (parte I)

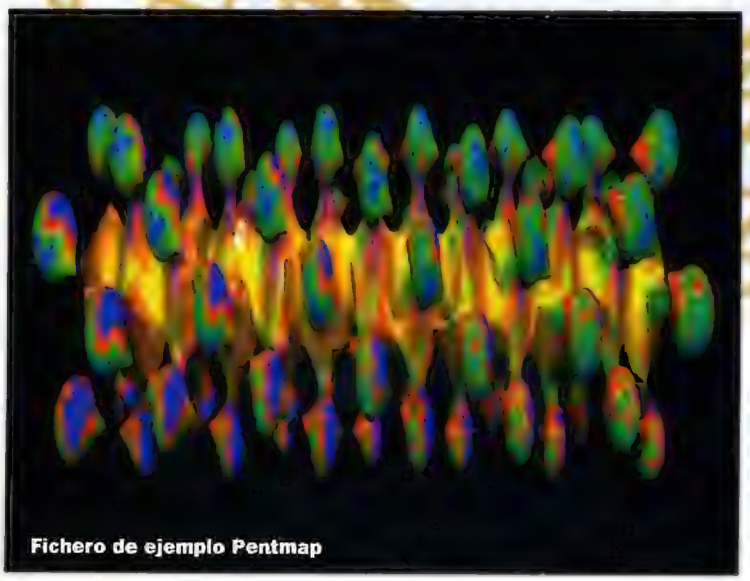
Fichero de ejemplo Sink

Crear animación por ordenador puede ser una tarea larga y tediosa, pero la recompensa merece la pena. A lo largo de los últimos meses se han recibido numerosas cartas de lectores que preguntaban cuándo se iba a tratar este tema. Coincidiendo con la presentación de la cuarta beta de «POV 3.1»,– este mes estudiaremos cómo pueden crearse películas de animación sintética con «POV».

Crear animaciones con «POV» es una tarea que presenta ciertas dificultades y ello se debe a que nuestro raytracer favorito no fue diseñado pensando en la creación de películas sintéticas. Para empezar, «POV» no dispone de un entorno integrado de ventanas en donde puedan definirse paths de movimiento para los modelos o la cámara. Además, «POV» no produce una salida en formato FLI, FLC o AVI para que podamos admirar la animación inmediatamente. De hecho, hasta la aparición de la versión 3.0, lo único que «POV» podía ofrecer para ayudar al sufrido creador de animaciones era una variable interna llamada clock (reloj), cuyo funcionamiento estudiaremos más adelante. Sin embargo, aun así, «POV» ha sido utilizado para generar animaciones casi desde su primera versión. Esto fue posible debido a que había muchos programadores entre los amantes de «POV» y por ello pronto empezaron a aparecer utilidades de todo tipo para apoyar a los creadores de animaciones.

La situación mejoró bastante con la aparición de la versión 3.0 de «POV», ya que dicha versión implementaba muchas sentencias de "programación" que podían ser empleadas en la generación de animaciones. Sin embargo, es posible que la verdadera revolución en este apartado se dé ahora, con la publicación de la versión 3.1 de «POV». ¿Por qué? Porque esta última versión aporta nuevas sentencias de "programación" que hacen innecesario el uso de algunas utilidades de apoyo que hasta ahora eran imprescindibles. En efecto, empleando sentencias como #if, #switch, array, #macro y otras, podremos programar ciertas animaciones que resultarían difíciles de generar incluso mediante el uso de programas con entornos visuales. El inconveniente, por supuesto, es que al tratarse de sentencias nuevas no hay, por ahora, ejemplos de creación de animaciones que empleen estas ventajas, ni tampoco demasiada documentación sobre el tema. (Lo mismo sucede con muchos pov-ipas que tendrán que ser actualizados para que puedan ser empleados con «POV 3.1»). Esto implica que tendremos que investigar muchas cosas por nosotros mismos si queremos prescindir del uso de conceptos y utilidades que en este momento ya se hallan anticuadas.

Todo esto no quiere decir, no obstante, que la animación con «POV» se haya convertido de repente en pan comido. Crear animaciones sigue siendo, con «POV» o cualquier otro programa, una tarea lenta y dura que en ocasiones puede calificarse de auténtico suplicio. Y si queremos conseguir buenos resultados el camino seguirá pasando –como siempre– por brutales exprimimientos neuronales, por docenas y docenas de pruebas, y por días, semanas e incluso meses de lentos cálculos para nuestras CPU. La única manera de minimizar nuestros esfuerzos estará en un sólido dominio de los conceptos básicos del funcionamiento de «POV» y en una cuidadosa planificación del trabajo antes de empezar a generar un solo frame.


Fichero de ejemplo Pentmap

Cifras básicas

Como todo el mundo sabe, cuando estamos viendo una película de imagen real o una animación hecha por ordenador, nuestro cerebro interpreta como un movimiento fluido algo que en realidad no es más que una secuencia de imágenes estáticas proyectadas una detrás de otra. Esto se debe a que nuestro sistema nervioso no es lo bastante rápido como para advertir el engaño.

Concretamente, para que percibamos esta sucesión de imágenes estáticas como una animación fluida, se necesita, al menos, una velocidad de 24 frames o cuadros por segundo, aunque muchas animaciones hechas por ordenador sólo exhiben unos 15 o 12 frames por segundo. (Nota: es de suponer que un extraterrestre, con una potencia mental tan notable como la de vulcaniano Mr. Spock, necesitará, al menos, una tasa de 300 frames por segundo para percibir una animación fluida).

Y ahora hagamos algunos números. Supongamos que queremos crear una animación que conste de unos 12 frames por segundo. Si deseamos generar una animación de un minuto de duración esto nos obligará a generar 12*60 = 720 escenas para montar la película. Suponiendo que estemos generando los frames a una resolución de 320*240 píxeles y asumiendo que utilicemos un formato donde se precise un byte por cada componente de color (por ejemplo, el TGA), nos dará como resultado una cifra de 230410 bytes para cada frame (en realidad hace falta algo más para guardar la cabecera del fichero). Por tanto, si tenemos 720 archivos de este tipo, necesitaremos unos 158 Megas sólo para almacenar la secuencia de frames a partir de la cual se va a construir la animación. Naturalmente esto no significa que dicha animación vaya a requerir la misma cantidad de memoria. Los formatos que suelen emplearse para almacenar las animaciones con las que vamos a trabajar pueden ser el FLI, el FLC y el AVI, y todos ellos emplean sistemas para ahorrar memoria (a costa de perder, más o menos, calidad gráfica).

Fichero de ejemplo Camera 2

Fichero de ejemplo Plotqit

Casi siempre se suelen procesar todos los cuadros de la animación para crear una paleta común que permita emplear un número de colores menor. Además, suele utilizarse también la animación diferencial. Este sistema se basa en la suposición de que la mayoría de los píxeles de un frame van a conservar su color con respecto al cuadro anterior, por lo que sólo será preciso almacenar los cambios con respecto a la escena del frame anterior. Esto puede hacer que sea prácticamente imposible predecir el tamaño final de un fichero de animación. Para comprobar esto se generaron tres archivos FLC a partir de tres ejemplos incluidos con la beta de «POV 3.1». Ya se ha dicho antes que actualmente «POV» no genera ficheros FLI, FLC o AVI, así que se tuvo que recurrir a una utilidad extema para realizar el montaje de la animación. Así, con el ya viejo Dta se generaron los FLC correspondientes a las animaciones de ejemplo slinky, camera y boing (en el directorio animate). Y el resultado fue que la memoria requerida para las animaciones se quedó en el 20%, el 10% y el 1% de lo demandado por sus secuencias de frames respectivas.

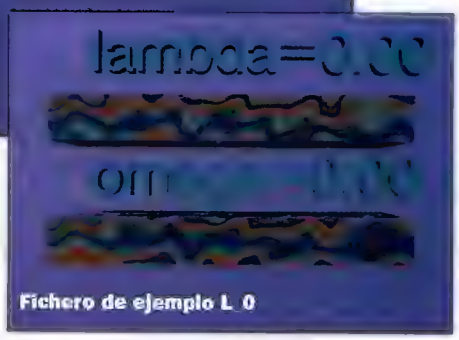
¿Y en cuanto al tiempo? Tampoco es fácil de calcular. Supongamos que tardamos tres minutos en generar uno de los frames de 320*240 del ejemplo anterior de 720 TGA. Entonces podríamos suponer que tardaríamos una media de 3*720 minutos en generar todos los frames de la escena, es decir, unas 36 horas. Pero realmente la cosa no es tan precisa. La animación puede estar compuesta por una serie de secuencias con planos diferentes y algunas de ellas pueden demandar mucho más tiempo de cálculo que otras. En este caso lo mejor será generar no una, sino varias animaciones –que serán montadas después en una sola– y efectuar los cálculos de tiempo en función del frame inicial de cada secuencia.

Por el momento estudiaremos cómo podemos manejar «POV» para crear las secuencias de frames a partir de las cuales se montarán las animaciones. Más adelante ya veremos qué utilidades pueden emplearse para realizar los mencionados montajes.

Conceptos básicos

Antes de crear una animación de cualquier tipo habremos de contar con los personajes que van a intervenir en ella. Dichos personajes pueden ser modelos creados desde el mismo «POV» o modelos importados construidos desde otros programas. Sin embargo, antes incluso de disponer de los protagonistas de la película, será conveniente contar con el guión de la misma. ¿Cuál es el plan? ¿Vamos a intentar narrar una historia por simple que sea? ¿O preferimos ilustrar una secuencia de efectos especiales? Sea cual sea la idea básica, como siempre será preferible utilizar papel y lápiz para asegurarnos de que lo planeado está lo suficientemente claro, ya que es mejor hallar los fallos en la fase de planificación que después, cuando hayamos generado 200 o 400 frames inservibles. La manera de preparar este guión dependerá un poco de cada autor. Quizá lo mejor sea preparar un guión muy esquemático de la película. Dicho guión podría estar compuesto por apartados numerados, siendo cada uno una descripción de un plano de cámara diferente al anterior. Además, cada uno de estos apartados incluiría también la enumeración de los cambios operados en los objetos y en sus texturas, así como en las fuentes de luz. Hecho esto podríamos preparar un storyboard dibujando una serie de viñetas abocetadas para cada nuevo plano, sólo para asegurarnos que tenemos una idea clara de cómo debe quedar visualmente la película. (Nota: no importa que no dibujemos demasiado bien. Lo importante es que queden bien ilustradas las posiciones de la cámara y los objetos).

Fichero de ejemplo Reflect


Fichero de ejemplo L_0

El siguiente paso será la preparación de los frames propiamente dichos. En muchos programas la creación de los frames se construye mediante frames clave. Cada frame clave contiene una descripción precisa de todos los objetos de la escena en un momento determinado del tiempo; su posición, orientación, texturas, etc. El método consiste en definir una serie de frames clave en donde cada objeto cambie en el tiempo según nuestro criterio. Posteriormente, el propio programa, a una orden nuestra, generará automáticamente todos los frames intermedios que deba haber entre los frames clave. A veces, cuando los cambios a definir sean muy sencillos, nos bastará con un par de frames clave para generar un montón de escenas intermedias. En otras ocasiones, en cambio, harán falta muchos más. La ventaja de este método, obviamente, radica en que es muy interactivo, ya que el usuario puede visualizar los cuadros clave cuando quiera (y por lo tanto puede hacerse una idea muy acertada de cómo va a ser aproximadamente la animación). Naturalmente se pueden cometer errores y puede hacerse necesario el insertar más frames clave o bien el tirarlos a todos a la basura y empezar de nuevo. Esta es la idea básica,

Órdenes para «POV» relativas a la generación de secuencias de frames

la orden en un fichero INI	la misma orden en la línea de comandos	función
Initial_Frame=n	+KFIn	Pone el frame inicial al valor n
Final_Frame=n	+KFFn	Pone el frame final al valor n
Initial_Clock=n.n	+KIn.n	Indica que el valor inicial del reloj sera n.n
Final_Clock=n.n	+KFn.n	Estipula que el valor final del reloj sera n.n

Figura 1

aunque cada programa incorpora sus propios refinamientos a este método básico de frames clave.

Explicado esto hay que aclarar que «POV» no funciona de esta manera. Con «POV», para cada animación independiente (aunque luego ésta pueda formar parte de otra mayor) sólo existe un frame clave: el inicial. Para generarlo escribiremos un fichero escénico utilizando el lenguaje de «POV», como lo haríamos para crear cualquier escena suelta. La diferencia estará en que ahora emplearemos una variable llamada clock que se utilizará como reloj de muchas o de todas las transformaciones descritas en la escena.

Órdenes para generar animaciones

Cuando invocamos a «POV» pasándole ciertos comandos, el programa entiende que debe generar no una única escena, sino una secuencia de ellas. Estos comandos se pueden impartir desde la línea de órdenes o bien escribiendo sentencias determinadas en un fichero de extensión INI. Los ficheros INI son archivos que guardan órdenes para «POV», y estas órdenes se ejecutan cuando se invoca al programa pasándole el nombre de un archivo INI como parámetro (sin que dicho nombre sea precedido por ningún símbolo ni letra). Los ficheros .INI constituyen, pues, una alternativa al sistema de línea de comandos, ya que existe una sentencia INI equivalente para cada comando posible de la línea de órdenes. Sin embargo, muchos pov-adeptos no utilizan ficheros INI. La razón de esto es que es superfluo aprender dos sistemas diferentes para conseguir un mismo resultado (y el sistema de la línea de órdenes es bastante más antiguo que el de los ficheros INI). Además, los comandos que podemos escribir en la línea de órdenes son más cortos y más sencillos de memorizar que las sentencias de los archivos INI, como podéis comprobar echando un vistazo a la **Figura 1**.

Con todo es conveniente que recordéis lo que es un fichero .INI porque muchos de los ejemplos de «POV» –entre los que se hallan los de animación– están almacenados en directorios donde también hay ficheros .INI utilizados para generar dichos ejemplos. En estos casos es típico que el nombre del fichero .INI sea el mismo que el del ejemplo .pov para el cual se guardan las especificaciones. Así, si echáis un vistazo al directorio "animate\boing", veréis que hay dos ficheros llamados boing1.pov y boing1.INI. Editando el archivo .INI, comprobaréis que éste imparte una serie de órdenes para «POV» pero no especifica la resolución de la imagen de salida. Por ello podemos renderizar boing1.pov, por ejemplo, con la orden de línea del **Cuadro 1**, lo que pasará a «POV» las órdenes escritas en el archivo denominado boing1.INI y añadirá también las dos que hemos indicado como comandos de línea (con lo que las imágenes tendrán una resolución de 320*240 pixels). Por otro lado, si estamos utilizando la versión «POV» para Windows, podremos hacer lo mismo editando el archivo INI y empleando la opción "Render/Edit Settings Render" para escribir las opciones +w320 y h240 en el apartado "comand line options". (Normalmente, con esta versión se utiliza esta opción o la de "render/ Start Rendering" para renderizar el archivo .POV que está siendo editado, pero también podemos hacer lo mismo con los ficheros .INI).

```
povray boing1 +w320 +h240
```

Cuadro 1

Bien, ahora que sabemos lo que son los ficheros .INI y cómo emplearlos para generar los ejemplos de animación, podemos continuar estudiando las órdenes de la **Figura 1**. En general, para ordenar la generación de una secuencia de imágenes bastará con emplear la orden +KFF. Veamos un posible caso. Supongamos que hemos indicado a «POV» que deseamos que el nombre del fichero de salida sea bola.tga. Pues bien, el comando +KFF100 ordenará al programa que renderice 100 escenas. El nombre de la primera será bola001.tga y el de la última, bola100.tga. Esto sucede porque, por defecto, el valor para el frame inicial (KFI) es 1. Si queremos que se generen 75 frames y que éstos vayan desde bola25.tga a bola100.tga, entonces deberemos añadir el comando +KFI25. (Nota: hay que tener cuidado con el nombre de la animación porque el número a añadir puede sobreescribirse sobre éste. Por ejemplo, si ordenamos 100 frames so-

Fichero de ejemplo Boing1

bre el nombre de salida mariposa.tga, entonces el primer fichero será marip001.tga y el último marip100.tga).

Sin embargo, cuando se emplea la orden +KFF, la generación de una serie de imágenes no es lo único que sucede. También ocurre que se activa una variable de coma flotante llamada clock, cuyo valor va cambiando, incrementándose automáticamente de manera proporcional para cada nuevo frame. Por defecto, el valor inicial de esta variable es 0 para la primera imagen de la serie y 1 para la última, independientemente de los valores dados a KFI y KFF. Así, y siguiendo con el ejemplo anterior, clock valdrá 0 para bola001.tga, 0.001 para bola002.tga y 1 para bola100.tga. Ahora bien, ¿para qué nos sirve la variable clock?

La variable clock

En «POV», el sistema para generar animación consiste en referenciar a la variable clock en las transformaciones espaciales de los objetos. Supongamos, por ejemplo, que tenemos una esfera situada en la posición X=0 y que deseamos hacer una animación de 100 frames en la que dicha esfera se desplace a la posición X=50. En tal caso emplearíamos la orden representada en el **Cuadro 2** dentro del fichero escénico:

```
sphere{<0,0,0>,10 pigment{Red}
translate<clock*50,0,0>}
```

Cuadro 2

En el primer frame, clock valdrá 0 y la esfera, por tanto, se renderizará en su posición de origen en el primer frame. En el siguiente, clock se incrementará al valor 0.001 y la esfera pasará a la posición 0.05, en el siguiente clock valdrá 0.002 y la esfera pasará a la posición 0.10, etcétera. Es fácil ver que la fórmula funciona correctamente porque el último valor de clock debe ser 1 y 1*50=50.

Ahora supongamos que 100 frames nos parecen demasiados para representar este sencillo movimiento y que queremos generar la misma animación con 50 frames. En este caso bastará con cambiar el valor de KFF por 50. En cuanto a la fórmula empleada en el ejemplo de antes, seguirá siendo válida porque clock se incrementa de 0 a 1 de manera proporcional al número de imágenes a generar. (Es decir, en el primer frame, clock valdrá 0, en el 25 valdrá 5 y en el último valdrá 1). Este sistema es muy cómodo porque podemos tirar pruebas de. por ejemplo, sólo 10 frames de una secuencia para ver si ésta es correcta y, luego, cuando hayamos comprobado su validez, podemos volver a renderizar la secuencia ordenando 50, 100 o 1000 frames.

¿Y qué sucede si, por ejemplo, queremos que las tgas a generar tengan números entre 25 y 125? En este caso utilizaríamos las órdenes de línea +KFI25 y +KFF125, pero la fórmula para emplear clock quedaría igual (si es que deseamos generar la misma secuencia del ejemplo anterior).

Ahora veamos otro ejemplo sencillo. La misma esfera de antes tiene aplicada una textura con el bitmap de la imagen de la tierra y deseamos que gire 360 grados en torno a su eje produciendo una animación compuesta por 20 frames. Pues bien, en este caso utilizaríamos la orden +KFF20 y escribiríamos la sentencia representada en el **Cuadro 3** en el fichero escénico.

```
sphere{<0,0,0>,10 texture{mundo}
rotate y*clock*360}
```

Cuadro 3

Por el momento se nos ha terminado el espacio permitido. Continuaremos el próximo mes.

Nota importante. Podéis remitirnos vuestros trabajos o consultas, bien por carta a la dirección que figura en el sumario de PCmanía, o vía e-mail a rendermania.pcmania@hobbypress.es

Animaciones a lo grande (Parte I)

A pesar de la advertencia sobre los problemas que comporta la publicación de animaciones de gran tamaño, este mes y el siguiente haremos una excepción con el fin de publicar algunos trabajos que nos fueron enviados antes de preveniros sobre el tema. Así, hallaréis aventuras estilo tron, cortos humorísticos, paseos por la nieve y otras cosas de interés.

Después de la paliza que los orcos les propinaron en el número 17, **Antonio Antón** ha decidido enviar refuerzos a nuestros caballeros. Para ello, Antonio ha creado un magnífico ejemplar de bárbaro y ha preparado también unas estupendas escenas donde se ilustra cómo sus hordas se disponen a acudir a la batalla. (Todo ello ha sido renderizado desde «POV», utilizando mesh). Para crear el modelo del bárbaro (que podéis hallar junto a los incs correspondientes en el CD-ROM), Antonio ha empleado –aparte de «POV»– los programas «Spatch», «Imagine 4» y «Hf-Lab».

Gracias por tus elogios a las escenas del número 17, las tuyas también son estupendas y de hecho tu personaje nos ha gustado tanto que hemos decidido darle un bando propio. Habrá, pues, batallas entre los caballeros del caos (nuestros), las hordas bárbaras (tuyas), la peste verde (los orcos) y las Evormigas. Por último, gracias por darnos a conocer la traducción del manual de «POV» que publicamos en este número (en tu directorio).

El Foro del Lector

Presentamos nuestras más abyectas disculpas a **Juan Pacheco Guerreiro** por haber dejado pasar tanto tiempo antes de publicar sus animaciones. Además, la primera parte de este trabajo no aparece este mes sino que lo hizo en el número anterior, en el apartado de animaciones. En fin, publicamos aquí la segunda parte del trabajo de Juan. En ella (como en la primera) se narran las aventuras de tres guerreros electrónicos en una rejilla de juegos y en lucha contra la cruel tiranía electrónica.

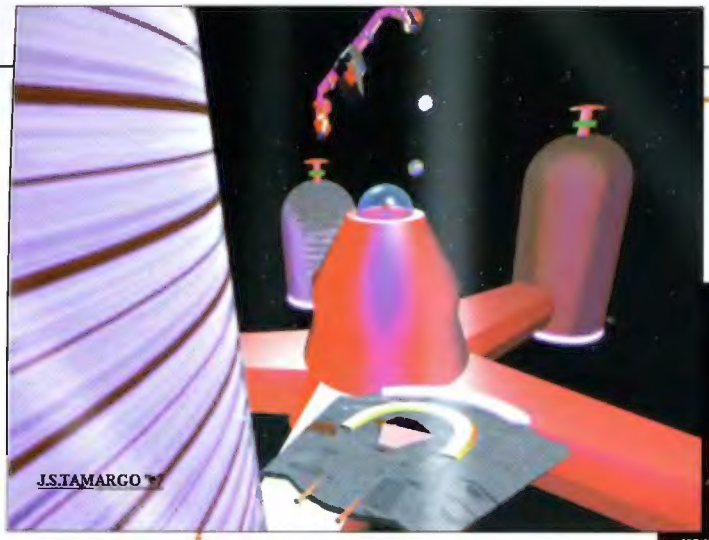
El trabajo de Juan está fuertemente basado en Tron, y de hecho gran parte de la música de fondo y las voces están tomadas de este clásico de Disney. A pesar de la sencillez de la geometría (imprescindible para generar los frames con rapidez), la ambientación está muy bien lograda y algunas escenas (sobre todo en la segunda parte) son prácticamente idénticas a las de la película. La expresividad de los tres héroes es extraordinaria y sus movimientos, así como los de la cámara, están muy bien hechos. La animación, además, no es una mera sucesión de imágenes, sino una historia bastante bien articulada que tiene escenas verdaderamente memorables. (Nuestras favoritas son la persecución de los tanques por la rejilla y la lucha final contra el gigante).

Sus únicos defectos radican en algunas escenas que quedan muy oscuras. Además, ambas partes "sólo" duran un total de unos 10 u 11 minutos.

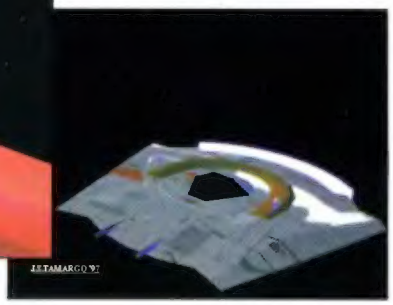
Nos hemos sorprendido al leer la carta de **Alfonso García Ordás** donde nos mencionaba que esta era la cuarta vez que enviaba la animación que publicamos este mes aquí, y cuya elaboración data del 96. Está claro que han surgido problemas de algún tipo porque tu paquete de discos estaba encima de la pila y no hemos encontrado ningún otro paquete similar con tu nombre.

En cuanto a tu animación, construida con «3D Studio», nos ha parecido fantástica. Tu película recuerda, por su desarrollo y por los gags, a un corto de dibujos animados de la Warner. Los movimientos de los personajes están muy cuidados y resultan muy expresivos.

¿Qué puede esperar quien envía una animación en 20 discos? Pues que algunos lleguen mal, claro. Esto es lo que ha sucedido con el trabajo de **Sergio Ruíz Navarro**, un amante de «3D Studio», «MAX» y «Rhino». Tan sólo ha sido posible rescatar la animación snow.fli (y poco más), donde disfrutaremos con las acrobáticas evoluciones de un deportista virtual sobre la nieve. Excelente animación del personaje y de la cámara y buen efecto manual de la nieve despedida por la tabla. El único problema está en que el tamaño del susodicho chorro de nieve permanece constante incluso en los derrapes, pero, claro, esto no es precisamente fácil de hacer. Sergio, bravo por la animación y por las escenas del coche.



El récord absoluto en tiempo de espera para animaciones perdidas en pozos espacio-temporales se lo lleva la carta de nuestro amigo **Javier Suárez Tamargo** cuyo trabajo, dos animaciones creadas con «3D Studio 4», publicamos ahora. ¡Un año de retraso! Bueno, tu carta debió caer detrás de una estantería accidentalmente. Esperamos que todavía leas estas páginas y puedas aceptar nuestras disculpas.



David Sánchez Beato Barrantes nos envía una animación para presentarnos a su grupo NeoDreams. Según afirma David, este grupo tiene en proyecto una animación en clave humorística de una batalla entre mechs que, calculan, ocupará unos 50 Megas comprimidos. Como estos autores saben lo difícil que nos sería publicar un trabajo de esta extensión, solicitan la creación de una sección fija o un concurso de animaciones. Se estudiará.

Jaime Garzón Pérez nos envía algunas imágenes creadas con «3D Studio 4» y pide crítica de dos de ellas. Pues bien, Bar0 queda bastante resultona salvo por la textura de la barra, que le da un relieve excesivo y las barras del techo que parecen excesivamente grandes. La puerta del bar también parece demasiado pequeña, aunque esto podría deberse a que la posición de la cámara no permite apreciar bien la geometría. De esta escena nos gusta el contraste de colores, la iluminación y las sombras resultantes.

En cuanto a Estación, nos parece una escena excelente por las texturas y la iluminación. Lo único que no nos ha gustado es la forma en que ha quedado la textura de la acera en el corte de ésta, ya que no está a la altura de todo lo demás.

Sergio Jorde Quintana pregunta si podemos enviarle los primeros 16 números de Rendermanía, sin PCmanía. Tememos que no. Ten en cuenta que en cada PCmanía viene un número de Rendermanía y no podemos extraer el suplemento sin dejar incompleta la revista. Pero, de todos modos, para iniciarte con «POV» no necesitas todos los números publicados de Rendermanía. Procura leerte los publicados desde el 16.

El Foro del Lector

Yago de Martín López nos envía unas interesantes escenas creadas con «3D Studio» entre las que destaca ¿un alienígena? montado en monopatín. Y respecto a la vieja controversia «POV» vs «3D Studio», también creemos que es demasiado simplista decir que un programa es mejor que el otro. Cada uno supera al otro en determinados puntos. Gracias por tu felicitación, tus escenas y tus mallas.

Diego Rodríguez Ibáñez nos envía una escena espacial realizada con el programa «POV» y otras en las que interviene un estupendo mech inspirado en un modelo de Battletech. Nuestro amigo Diego espera que su mech, realizado con «3D Studio», participe en la próxima escena de mechs. Puedes contar con ello.

Jose Milagros García Lorenzo ha decidido tomar parte por el bando de las evormigas en las inminentes confrontaciones que tendrán lugar entre esta especie y todas las demás, en nuestro pequeño mundo medieval. Para ello, José ha enviado las mallas de un escarabajo, una mantis y una libélula. El escarabajo será utilizado como un elefante de guerra (por su tamaño), la mantis será utilizada para la "caballería" de las evormigas y la libélula será empleada para aumentar el radio de vuelo de las Evormigas. Esto quiere decir que los demás bandos (orcos, caballeros y bárbaros) pueden haber quedado en cierta inferioridad pues, aunque disponen de artillería, carecen de caballería y fuerzas aéreas veloces. ¡Hay que seguir con la carrera armamentística! En cuanto a las criaturas de José, las emplearemos en la próxima batalla donde participen las evormigas, pero no publicaremos las mallas puesto que así lo ha solicitado el autor. Gracias por tu ayuda a las fuerzas insectoides.